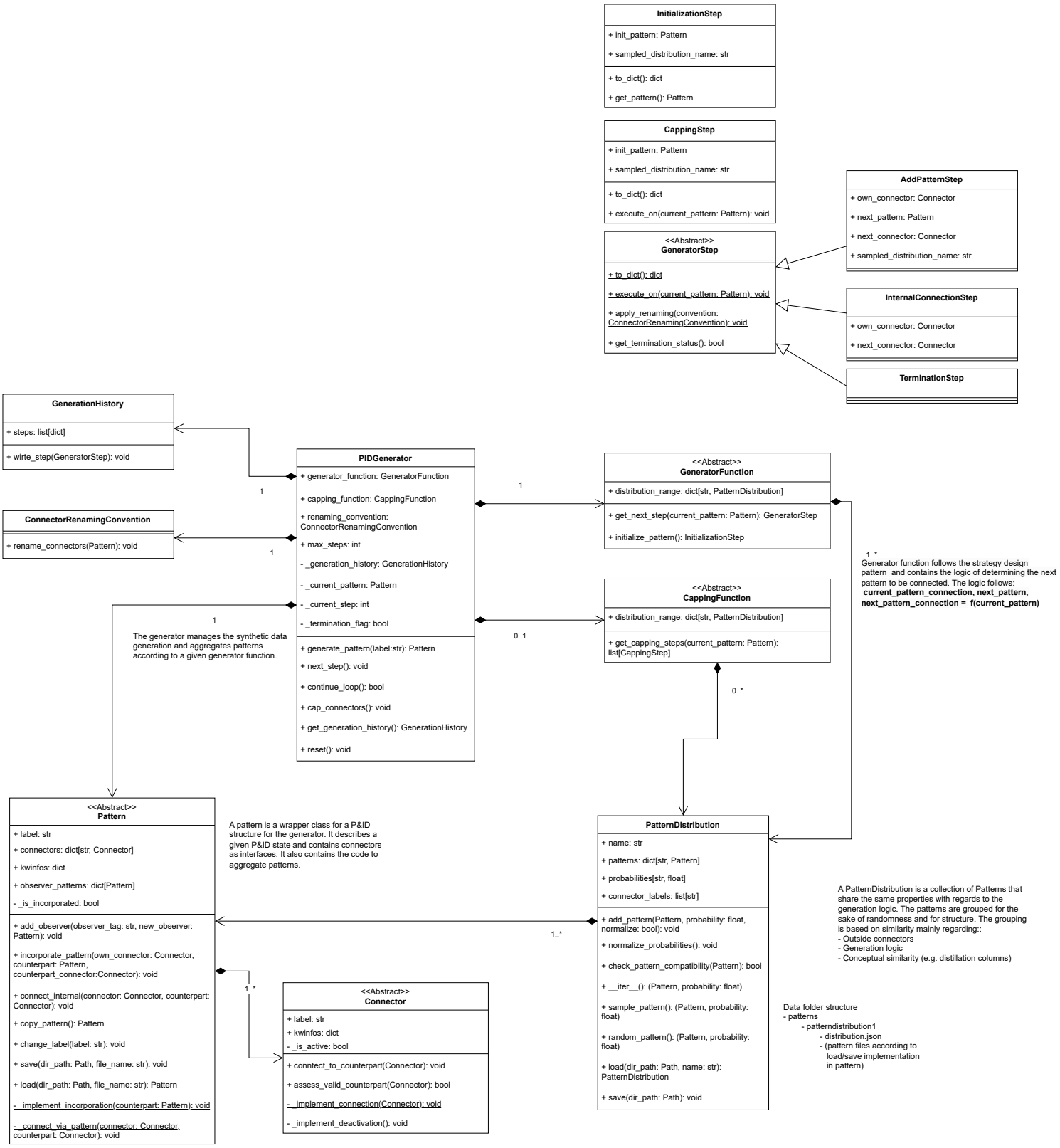
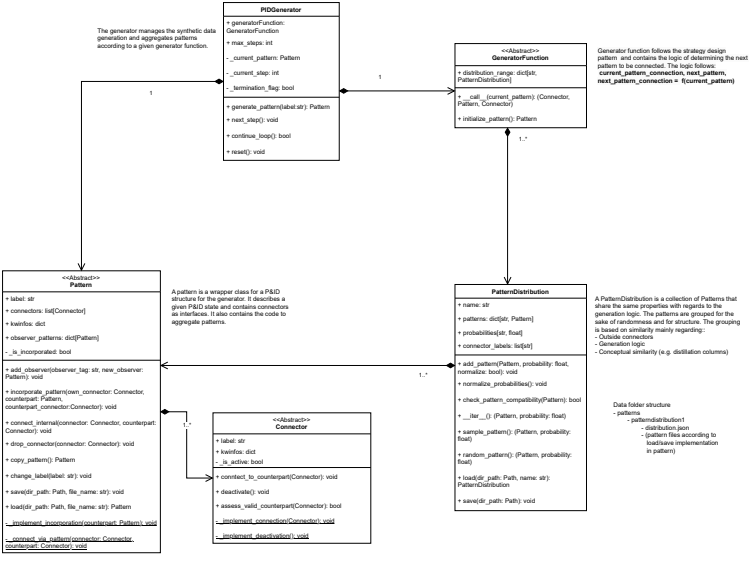
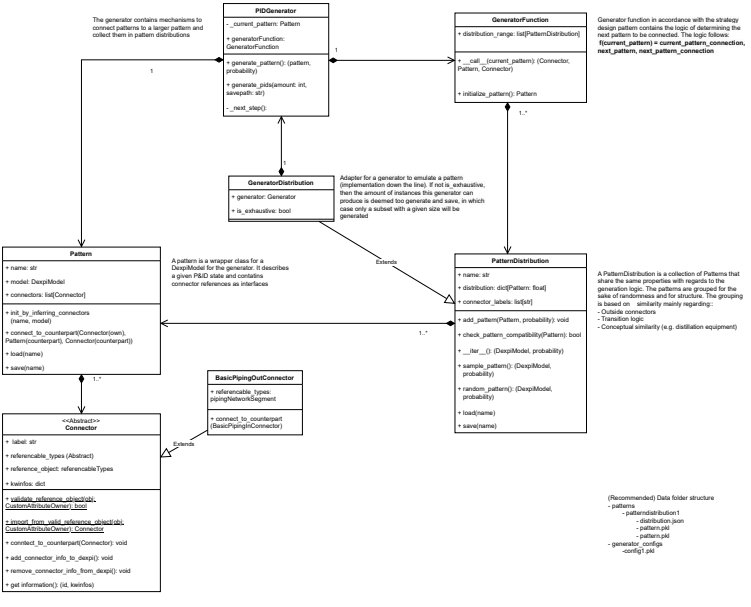

InitializationStep
+ init_pattern: Pattern
+ sampled_distribution_name: str
+ to_dict(): dict
+ get_pattern(): Pattern
CappingStep
+ init_pattern: Pattern
+ sampled_distribution_name: str
+ to_dict(): dict
+ execute_on(current_pattern: Pattern): void
AddPatternStep
+ own_connector: Connector
+ next_pattern: Pattern
+ next_connector: Connector
+ sampled_distribution_name: str
<<Abstract>>
GeneratorStep
+ to_dict(): dict
+ execute_on(current_pattern: Pattern): void
+ apply_renaming(convention: ConnectorRenamingConvention): void
+ get_termination_status(): bool
InternalConnectionStep
+ own_connector: Connector
+ next_connector: Connector
TerminationStep
GenerationHistory
+ steps: list[dict]
+ wirte_step(GeneratorStep): void
ConnectorRenamingConvention
+ rename_connectors(Pattern): void
PIDGenerator
+ generator_function: GeneratorFunction
+ capping_function: CappingFunction
+ renaming_convention: ConnectorRenamingConvention
+ max_steps: int
- _generation_history: GenerationHistory
- _current_pattern: Pattern
- _current_step: int
- _termination_flag: bool
+ generate_pattern(label:str): Pattern
+ next_step(): void
+ continue_loop(): bool
+ cap_connectors(): void
+ get_generation_history(): GenerationHistory
+ reset(): void
1
1
1
1
<<Abstract>>
GeneratorFunction
+ distribution_range: dict[str, PatternDistribution]
+ get_next_step(current_pattern: Pattern): GeneratorStep
+ initialize_pattern(): InitializationStep
<<Abstract>>
CappingFunction
+ distribution_range: dict[str, PatternDistribution]
+ get_capping_steps(current_pattern: Pattern): list[CappingStep]
0..1
0..*
1..*
Generator function follows the strategy design pattern and contains the logic of determining the next pattern to be connected. The logic follows:
current_pattern_connection, next_pattern, next_pattern_connection = f(current_pattern)
The generator manages the synthetic data generation and aggregates patterns according to a given generator function.
<<Abstract>>
Pattern
+ label: str
+ connectors: dict[str, Connector]
+ kwinfos: dict
+ observer_patterns: dict[Pattern]
- _is_incorporated: bool
+ add_observer(observer_tag: str, new_observer: Pattern): void
+ incorporate_pattern(own_connector: Connector, counterpart: Pattern, counterpart_connector:Connector): void
+ connect_internal(connector: Connector, counterpart: Connector): void
+ copy_pattern(): Pattern
+ change_label(label: str): void
+ save(dir_path: Path, file_name: str): void
+ load(dir_path: Path, file_name: str): Pattern
- _implement_incorporation(counterpart: Pattern): void
- _connect_via_pattern(connector: Connector, counterpart: Connector): void
A pattern is a wrapper class for a P&ID structure for the generator. It describes a given P&ID state and contains connectors as interfaces. It also contains the code to aggregate patterns.
1..*
<<Abstract>>
Connector
+ label: str
+ kwinfos: dict
- _is_active: bool
+ conntect_to_counterpart(Connector): void
+ assess_valid_counterpart(Connector): bool
- _implement_connection(Connector): void
- _implement_deactivation(): void
PatternDistribution
+ name: str
+ patterns: dict[str, Pattern]
+ probabilities[str, float]
+ connector_labels: list[str]
+ add_pattern(Pattern, probability: float, normalize: bool): void
+ normalize_probabilities(): void
+ check_pattern_compatibility(Pattern): bool
+ __iter__(): (Pattern, probability: float)
+ sample_pattern(): (Pattern, probability: float)
+ random_pattern(): (Pattern, probability: float)
+ load(dir_path: Path, name: str): PatternDistribution
+ save(dir_path: Path): void
1..*
A PatternDistribution is a collection of Patterns that share the same properties with regards to the generation logic. The patterns are grouped for the sake of randomness and for structure. The grouping is based on similarity mainly regarding::
- Outside connectors
- Generation logic
- Conceptual similarity (e.g. distillation columns)
Data folder structure
- patterns
- patterndistribution1
- distribution.json
- (pattern files according to load/save implementation in pattern)


PIDGenerator
+ generatorFunction: GeneratorFunction
+ max_steps: int
_ current_pattern: Pattern
_ current_step: int
_ termination_flag: bool
+ generate_pattern(label:str): Pattern
+ next_step(): void
+ continue_loop(): bool
+ reset(): void
The generator manages the synthetic data generation and aggregates patterns according to a given generator function.
1
1
<<Abstract>>
GeneratorFunction
+ distribution_range: dict[str, PatternDistribution]
+ __call__(current_pattern): (Connector, Pattern, Connector)
+ initialize_pattern(): Pattern
Generator function follows the strategy design pattern and contains the logic of determining the next pattern to be connected. The logic follows:
current_pattern_connection, next_pattern, next_pattern_connection = f(current_pattern)
1..*
<<Abstract>>
Pattern
+ label: str
+ connectors: list[Connector]
+ kwinfos: dict
+ observer_patterns: dict[Pattern]
_ is_incorporated: bool
+ add_observer(observer_tag: str, new_observer: Pattern): void
+ incorporate_pattern(own_connector: Connector, counterpart: Pattern, counterpart_connector:Connector): void
+ connect_internal(connector: Connector, counterpart: Connector): void
+ drop_connector(connector: Connector): void
+ copy_pattern(): Pattern
+ change_label(label: str): void
+ save(dir_path: Path, file_name: str): void
+ load(dir_path: Path, file_name: str): Pattern
_ _implement_incorporation(counterpart: Pattern): void
_ _connect_via_pattern(connector: Connector, counterpart: Connector): void
A pattern is a wrapper class for a P&ID structure for the generator. It describes a given P&ID state and contains connectors as interfaces. It also contains the code to aggregate patterns.
1..*
1..*
<<Abstract>>
Connector
+ label: str
+ kwinfos: dict
_ is_active: bool
+ connect_to_counterpart(Connector): void
+ deactivate(): void
+ assess_valid_counterpart(Connector): bool
_ _implement_connection(Connector): void
_ _implement_deactivation(): void
PatternDistribution
+ name: str
+ patterns: dict[str, Pattern]
+ probabilities[str, float]
+ connector_labels: list[str]
+ add_pattern(Pattern, probability: float, normalize: bool): void
+ normalize_probabilities(): void
+ check_pattern_compatibility(Pattern): bool
+ __iter__(): (Pattern, probability: float)
+ sample_pattern(): (Pattern, probability: float)
+ random_pattern(): (Pattern, probability: float)
+ load(dir_path: Path, name: str): PatternDistribution
+ save(dir_path: Path): void
A PatternDistribution is a collection of Patterns that share the same properties with regards to the generation logic. The patterns are grouped for the sake of randomness and for structure. The grouping is based on similarity mainly regarding:
- Outside connectors
- Generation logic
- Conceptual similarity (e.g. distillation columns)
Data folder structure
- patterns
- patternsdistribution1
- distribution.json
- (pattern files according to load/save implementation in pattern)


The generator contains mechanisms to connect patterns to a larger pattern and collect them in pattern distributions
PIDGenerator
- _current_pattern: Pattern
+ generatorFunction: GeneratorFunction
+ generate_pattern(): (pattern, probability)
+ generate_pids(amount: int, savepath: str)
- _next_step():
1
1
1
GeneratorFunction
+ distribution_range: list[PatternDistribution]
+ __call__(current_pattern): (Connector, Pattern, Connector)
+ initialize_pattern(): Pattern
Generator function in accordance with the strategy design pattern contains the logic of determining the next pattern to be connected. The logic follows:
f(current_pattern) = current_pattern_connection, next_pattern, next_pattern_connection
1..*
1
GeneratorDistribution
+ generator: Generator
+ is_exhaustive: bool
Adapter for a generator to emulate a pattern (implementation down the line). If not is_exhaustive, then the amount of instances this generator can produce is deemed too generate and save, in which case only a subset with a given size will be generated
Extends
Pattern
+ name: str
+ model: DexpiModel
+ connectors: list[Connector]
+ init_by_inferring_connectors (name, model)
+ connect_to_counterpart(Connector(own), Pattern(counterpart), Connector(counterpart))
+ load(name)
+ save(name)
A pattern is a wrapper class for a DexpiModel for the generator. It describes a given P&ID state and contains connector references as interfaces
PatternDistribution
+ name: str
+ distribution: dict[Pattern: float]
+ connector_labels: list[str]
+ add_pattern(Pattern, probability): void
+ check_pattern_compatibility(Pattern): bool
+ __iter__(): (DexpiModel, probability)
+ sample_pattern(): (DexpiModel, probability)
+ random_pattern(): (DexpiModel, probability)
+ load(name)
+ save(name)
A PatternDistribution is a collection of Patterns that share the same properties with regards to the generation logic. The patterns are grouped for the sake of randomness and for structure. The grouping is based on similarity mainly regarding:
- Outside connectors
- Transition logic
- Conceptual similarity (e.g. distillation equipment)
1..*
1..*
BasicPipingOutConnector
+ referencable_types: pipingNetworkSegment
+ connect_to_counterpart (BasicPipingInConnector)
Extends
<<Abstract>>
Connector
+ label: str
+ referencable_types (Abstract)
+ reference_object: referencableTypes
+ kwinfos: dict
+ validate_reference_object(obj, CustomAttributeOwner): bool
+ import_from_valid_reference_object(obj, CustomAttributeOwner): Connector
+ connect_to_counterpart(Connector): void
+ add_connector_info_to_dexpi(): void
+ remove_connector_info_from_dexpi(): void
+ get information(): (id, kwinfos)
(Recommended) Data folder structure
- patterns
- patternidistribution1
- distribution.json
- pattern.pkl
- pattern.pkl
- generator_configs
-config1.pkl